

École polytechnique de Louvain

Binarized Neural Networks

Bat call classification

Author: **Nathan DE WINTER**

Supervisor: **Siegfried NIJSSEN**

Readers: **Lucile DIERCKX, Maxime PIRAUX**

Academic year 2021–2022

Master [120] in Data Science: Information technology

Contents

1	Introduction	8
2	Concept Definition	11
2.1	Neural Network	11
2.1.1	Convolution Layer	12
2.1.2	Batch Normalization Layer (BN)	13
2.1.3	Dropout layers	14
2.2	Larq	14
2.3	Quantization function	16
2.4	Binary Neural Network	16
2.5	XGBoost	18
2.6	Non Maximum Suppression (NMS)	18
3	Previous Works	20
3.1	Detection	20
3.2	Multi class classification	20
3.3	Multi label classification	21
4	Datasets descriptions	23
5	Methods for binarizing neural networks	27
5.1	Network quantization after training	27
5.2	Network quantization during training	29
6	Comparisons with previous architectures	31
6.1	Batmen	31
6.2	Double CNN	32
6.3	Hybrid model with XGBoost	34
6.4	ResNet	35
6.5	Overall Comparison	37
7	Architecture optimization for binarized network	38
7.1	Moving BatchNorm layers	38
7.2	Results of the new architecture	40

8	Reducing XGBoost model size in memory	41
8.1	Number of estimators	41
8.2	Maximum depth	43
8.3	Minimum child weight	45
8.4	Choosing a model	47
9	Executing the final model on a Raspberry Pi	51
10	Conclusion	54
11	References	55
12	Hardware and Software Specifications	58
12.1	Software requirements	58
12.2	Hardware specifications	59
12.2.1	Computer for training the models	59
12.2.2	Raspberry Pi 3B	59
12.2.3	Raspberry Pi 400	59
12.2.4	Raspberry Pi Zero 2w	60
13	Appendices	61
13.1	Hyperparameters of each model	61
13.1.1	Batmen	61
13.1.2	CNN2	62
13.1.3	Hybrid CNN XGBoost	64
13.2	Inference times of the Raspberry Pi 400 on a 110 seconds file	65
13.3	Inference times of a Raspberry Pi Zero 2w on a 3 seconds file	66
13.4	Execution time of the XGBoost model with 500 estimators	67
13.5	Metrics Definition	67
13.5.1	Precision	67
13.5.2	Recall	68
13.6	Problems encountered	68
13.6.1	Error when training XGBoost model with large batch size	68
13.6.2	Building Larq Compute Engine on Raspberry Pi	68

Abbreviations

BN Batch Normalization.

BNN Binary Neural Network.

CNN Convolutional Neural Network.

DPNN Double Precision Neural Network.

FN False Negative.

FP False Positive.

GB Gigabyte.

KB Kilobyte.

MB Megabyte.

NMS Non Maximum Suppression.

OOM Out Of Memory.

RAM Random Access Memory.

ResNet Residual Neural Network.

STE Straight Through Estimator.

TN True Negative.

TP True Positive.

XGBoost Extreme Gradient Boosting.

List of Figures

2.1	Neural unit representation	12
2.2	Convolution process	13
3.1	Description of the revised Batmen architecture from previous thesis	21
5.1	First version of the revised Batmen architecture	27
6.1	Architecture of Batmen for training BNNs	31
6.2	Architecture of CNN2 for BNNs	33
6.3	Architecture of the hybrid XGBoost model for BNNs	34
7.1	Visualization of the non optimized architecture	39
7.2	Visualization of the optimized architecture	39
8.1	Evolution of the model sized compared to the number of estimators	42
8.2	Relation between the number of estimators and the different precision values	43
8.3	Evolution of the model sized compared to the maximum depth . . .	44
8.4	Relation between the maximum depth and the different precision values	45
8.5	Evolution of the model sized compared to the minimum child weight	46
8.6	Relation between the minimum child weight and the different precision values	47
8.7	Comparison of different models between their size and their precision for the detection task	48
8.8	Comparison of different models between their size and their global precision	49

List of Tables

4.1	Number of files from each database used for training and testing. . .	24
4.2	Number of calls from each database used for training and testing. . .	24
4.3	Proportions of calls for each label	25
5.1	Detection results post quantization	28
5.2	Classification results post quantization	28
5.3	Detection results after training	29
5.4	Classification results after training	29
6.1	Performance of Batmen	32
6.2	Performances of CNN2	33
6.3	Performances of the hybrid XGBoost model	35
6.4	Architecture of ResNet	36
6.5	Performances of ResNet	36
6.6	Performances comparison	37
7.1	Performances comparison between the non optimized and the opti- mized architecture	40
8.1	Performance comparison of the final model	50
9.1	Benchmarks DPNN on Raspberry Pi	51
9.2	Benchmarks BNN on Raspberry Pi	52
13.1	Hyperparameters of the Batmen model.	61
13.2	Hyperparameters of the CNN2 model (detection)	62
13.3	Hyperparameters of the CNN2 model (classification)	63
13.4	Hyperparameters of the hybrid XGBoost model.	64
13.5	Benchmark DPNN with a large file	65
13.6	Benchmark BNN with a large file	66
13.7	Benchmark DPNN Raspberry Pi Zero 2w	66
13.8	Benchmark BNN Raspberry Pi Zero 2w	67

13.9 Benchmark using XGBoost model with 500 estimators	67
--	----

Acknowledgements

I would like to thanks several people that helped me during the preparation of this thesis.

I would like to thank Siegfried Nijssen for his good advice and the discussions that we had during our meetings. Thanks also to his teaching method that I always find the most suitable in my case.

I would also like to thank Lucile Dierckx for her great help during this year and for the wonderful work that she has done with Mélanie Beauvois last year.

Thanks also to Maxime Piraux for giving me access to the Gotham server in order to access the Natagora database. His help for resolving some technical problems concerning me was appreciated.

Thanks to Natagora and the Plecotus team for sharing a large collection of recordings in order to permit us to always train better models for classifying bat calls.

Thanks also to Maxime Franco and Corrado Lipani for their work in the past years and their preprocessing on the Natagora database.

And thanks to my family for their support during these last years even if it was not easy every day.

Chapter 1

Introduction

In this thesis, we will explore the problem of multi class classification of bat species by analyzing their calls. Analyzing bat calls is important to measure their populations because some species present in Belgium are endangered mainly due to human activities. Automated tools are important for classifying these calls because doing it by hand is a long and complex task which can only be achieved by professionals. There exist some tools for solving this problem but some of them are commercial such as Sonochiro. Therefore, we need to develop open-source tools in order to make them accessible to a larger public and thus simplify the census of the different bat species populations.

Previous approaches were explored using Convolutional neural networks for detecting and classifying the calls according on the species that emit them. Nevertheless, the neural networks used in these previous works are computationally demanding since they use double-precision values as weights and outputs. There are many reasons for reducing the computational power demanded by these networks. An example is that we might want to run these networks on low-computing power computers for classifying the recorded calls directly in the field. But to make this possible, we will certainly use batteries and by reducing the complexity of the calculations, we will use less energy and the battery will last longer. This type of problem is also important in a green computing context since we want to reduce our footprint on the environment. By reducing the impact of our energy consumption, we might be able to better protect species endangered by our activities, such as bats.

Thus, the new approach explored in this thesis is about Binarized neural networks which only use 1 bit weights and output values. This type of network is known for being less computationally demanding which means that they can even be executed on low computing-power computers. But using less bits to represent the weights

and the outputs decrease the performances of these networks. Thus, we will observe if we can use Binarized neural networks for classifying bat calls while discussing about different topics in order to create a good architecture. In addition, we will search to reduce the size of an XGBoost model while keeping good performances. XGBoost models were also used in previous works and they showed to obtain the best performances when combined with a neural network used for classifying bat calls.

Concerning the different chapters, in Chapter 2, we will discuss about the main theoretical concepts in order to understand the principle of Binarized neural networks. In Chapter 3, we will discuss about previous works which aimed to solve the problem about detecting and classifying bat calls. In Chapter 4, I will present the different datasets used to train the different models. In Chapter 5, as a first approach to binarize neural networks, we will observe the effects of different binarization techniques in order to obtain binarized neural networks. I will compare two of them in order to determine which one is the most suitable to obtain the best results.

In Chapter 6, I will try to train such models while trying to reduce the loss induced by the binarization process by creating a first modified version of each architecture from previous works. Indeed, since these networks are using 1 bit weights and outputs, it is a matter of facts that they will perform poorly compared to their 32 bits counterparts. But still, there exists some techniques to reduce this problem by using some type of layers inside their architecture [1], [2]. In Chapter 7, we will discuss about a second version of the binary neural network using an XGBoost model. This new version aims to profit from operations fusing done by the Larq Compute Engine library [3]. Indeed, some operations can be fused as one for reducing the number of operations for executing the model. I will search to optimize this technique in order to minimize the model size and the execution time.

In Chapter 8, we will also try to understand how we can reduce the size in memory of machine learning models such as XGBoost in order to find an optimal size while keeping good performances. This size reduction is also important since the goal is to try to run them on low-computing power computers. Another objective of this chapter is to reduce the execution time of this model.

The last point that I will discuss in Chapter 9, is the performance of a model running on different Raspberry Pi devices. We will observe the advantages of using Binary neural networks compared to the classical neural networks. I talked about

their size in memory which is much lower than their classical counterparts but they are also faster for inferring on new data.

Chapter 2

Concept Definition

Before we can get to the main points, we need to talk about some theoretical aspects in order to understand the particularities of Binary neural networks.

2.1 Neural Network

Before we discuss about Binary neural networks, I will explain the concept the concept of neural network. A neural network functions by propagating information through the neurons composing it in a similar way than in a biological brain. The difference is that these neurons or units are represented as functions when used inside a layer of a given neural network. A layer is a composition of several units which takes an input from a previous layer and gives an output as result. The input from the previous layer is multiplied by a corresponding weight in order to obtain the output. But this only works inside the intermediate layers, also known as hidden layers, and the output layer of the whole network since the input layer takes only the raw input without applying any weight on it. The units of these hidden layers are also called hidden units. We can see the connections of a given hidden unit as follow in Figure 2.1.

Therefore, the output of such units can be formulated as follows:

$$Y_{i,l+1} = \sum_{j=1}^n w_{j,l} * X_{j,l} + w_{0,l}$$

where $X_{j,l}$ is the input value from unit j in layer l ,

$w_{j,l}$ is the weight associated to the unit j in layer l ,

$w_{0,l}$ corresponds to the bias of the layer l and

$Y_{i,l+1}$ is the output of unit j in layer $l+1$.

n is the number of units contained in layer l .

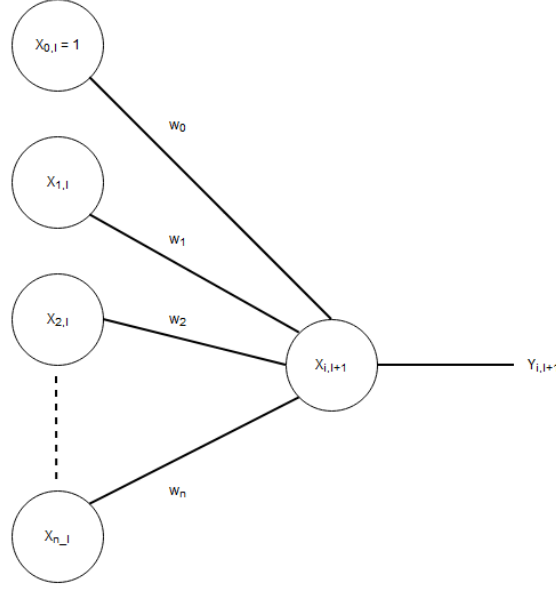


Figure 2.1: Representation of a unit inside a layer $l + 1$ which takes as input the different outputs from the previous layer l of units. The parameters w_i corresponds to the weights applied to the outputs of units $X_{i,l}$. The unit $X_{0,l}$ corresponds to the bias, thus, its output values is always 1 which means that only the weight w_0 defines the importance of the bias.

The formula representing the activation function can be applied to all units from each layer with the exception of the input layer which does not depends from previous units. When this formula is indeed applied to a given layer, we talk about a Dense layer or Fully Connected layer because all of its units used the outputs from the previous layer units.

There exist several types of layers aside from the Dense layer such as the Convolution layer or layers with a more specific role such as Batch Normalization layers or Dropout layers. These layers are also used in this work, thus, I will explain them in the following subsections.

2.1.1 Convolution Layer

The Convolution layer aims to extract features from the input by reducing its size. This kind of layers uses filters in order to compute an output from different parts of the input. This process is comparable to the one of the Dense layer but the difference is that the we apply the same weights on the different parts of the input. This process can be then applied on multidimensional data such as images or time

series. The computation from the filters can explained as follow in Figure 2.2.

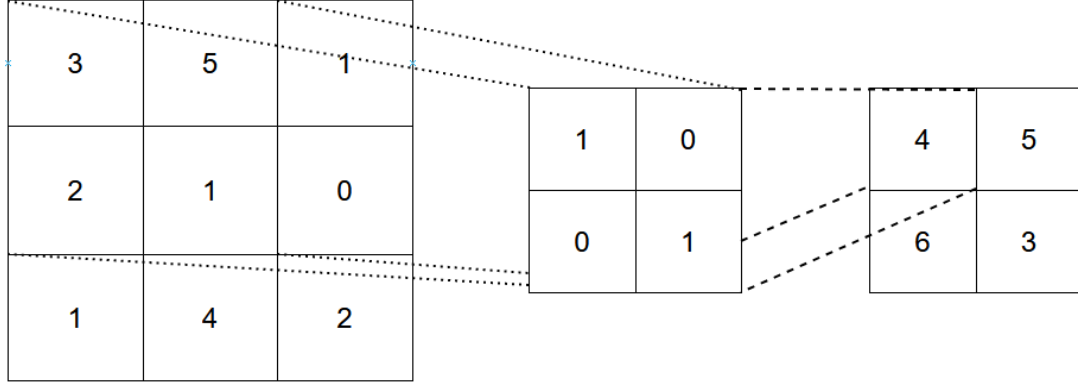


Figure 2.2: Convolution process where the filter is applied on part of the input. The result is 4 is computed as follows: $(3 * 1) + (5 * 0) + (2 * 0) + (1 * 1) = 4$. The other resulting outputs are computed with the same logic.

As you can see from Figure 2.2, the Convolution layer is well suited for extracting specific data from its input. In this case, it works on an image type input but it can be applied on 3D objects or more.

2.1.2 Batch Normalization Layer (BN)

This type of layer only applies a linear transformation to normalize its input. It can be represented by these equations:

$$y_i = \gamma \hat{x}_i + \beta$$

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

$$\mu_B = \frac{1}{n} \sum_{i=1}^n x_i$$

$$\sigma_B^2 = \frac{1}{n-1} \sum_{i=1}^n (x_i - \mu_B)^2$$

This is the same equation as defined in the Paper [4]. The normalization is done through the definition of the normal distribution. The mean μ_B and the variance σ_B^2 are computed during the training over each batch of data. The parameters γ and β need also to be learned by the network. At inference time, these two

parameters are fixed depending on their value learned by the network. This is due to the original intuition behind the BN layers since these parameters, as well as the mean and the variance, will be the same on new data. This intuition is well known in statistics when the distribution of a large sample is more or less equivalent when compared to the whole population from which it was retrieved. In our case, the totality of the different mini batches used during training represent the large sample that we need. The constant ϵ can be chosen or ignored depending on the choice of design. This layer is helpful during training because it helps to regularize the outputs of the layers inside a network by normalizing it for the next layer which takes these outputs as inputs this time. This normalization is important for keeping the output values inside a certain range centered on 0 which is useful for a Binarized neural network since the range of possible values is between -1 and 1. BN layers are also useful since they tend to reduce the loss of a neural network.

2.1.3 Dropout layers

This type of layer is simple to understand because it only deactivates some units of a layer randomly during training. It tends to improve the performance of a neural network by making the different units independent from each other. Therefore, it helps to reduce overfitting, a phenomenon where the network only memorizes the training examples, and thus, the model is more robust when confronted to new data. Even if they are commonly used for training neural networks, I will not use further this type of layer inside the architectures of the Binary neural networks. It is due by the fact that the binarization process can be seen as a variant to a Dropout layer [1] and thus adding Dropout layers to Binarized networks does not improve more the performance of such models. Even worse, using a high drop probability of 0.5 or higher, as usually used for neural networks with double-precision weights and outputs, can worsen the performance of a binary neural network. Thus, only probabilities smaller than 0.3 or 0.2 can be used but the impact of such layers is then too small on the performance. But their utility in BNNs is more questionable since their output is 1 when a unit is dropped and not 0 due to the STE activation function. This is why I will not use them further in this thesis but they are important to introduce since they were indeed used in previous architectures.

2.2 Larq

As binary networks are important, they have been implemented in an existing library, which is called Larq [5]. Larq is a python library whose aim is to create and to train binary neural networks. This library was constructed on top of Tensorflow [6] which is another library well known for creating and training neural networks.

Larq implements new versions of classical layers such as Dense layers or Convolution layers, these new versions are known as Quantized layers. Therefore, we can find different Quantized layers based on already existing layers. The difference with these Quantized layers, is that they use Quantization functions but I will introduce them with more precision in the next Section, the only thing to know now about these functions is that they are used to binarize the weights and the outputs of the different layers inside the network.

Larq is also composed of two subsidiary libraries called Larq-zoo [7] and Larq Compute Engine [3]. Larq-zoo is a collection of different famous neural network architectures in their binarized version. We can find for example ResNet [8] which is a famous architecture who has a lot of shortcuts. ResNet uses these skips in order to train on the residuals of the outputs from previous layers. This method helps the network to avoid vanishing and exploding gradients which helps to create large architecture with a lot of layers. We will discuss about its precise architecture in a further Section.

Larq Compute Engine is the second library whose purpose is to convert binary neural network model to a FlatBuffer format in order to execute them on low-computing power computers. The FlatBuffer format was used first in Tensorflow Lite for representing neural network as buffers and not as a graph which is the case during the training process. Thus, these two libraries convert trained models in this new format in order to use them only for inference. But the particularity of Larq Compute Engine, is that it is optimized for Binary networks. Therefore, it will store the weights using only 1 bit for each, in comparison, Tensorflow Lite stores them as 32 bits values but it can still decrease the number of bits to a minimum of 8 if it is specified. These two libraries search to fuse operations in order to reduce the number of operations which reduce the execution time but also the model size in memory. Fusing two operations is only possible when these two are compatibles, indeed, the Convolution process can be fused with a BN layer since the formula of the BN layer can directly be applied on the output of the Convolution layer. It is feasible because the multiplier γ of the BN layer can directly be applied on the weights and the term β on the bias. It is not feasible for example with a MaxPool layer because it only calculates the maximum values from different parts of its input, therefore, the operations of a BN layer can only be applied after this operation. Even if double-precision values are applied on the binarized weights, the calculations remain simpler than in a neural network using double-precision values since the number of this type of parameters is much lower.

2.3 Quantization function

Now that we have seen the different components of neural networks and that we have discussed about Larq, it is important to talk more precisely about the quantization function, which is really important for defining a Binary neural network. A quantization function is used to convert a value to a corresponding one by reducing the number of bits originally used for representing it. In our case, the quantization function will be used to represent numbers on 1 bit.

The most used quantization function for defining Binary neural networks is the Straight Through Estimators (STE). The output of this function gives only -1 or 1 depending on the input being a positive or a negative number:

$$q(x) = \begin{cases} 1 & \text{when } x \geq 0 \\ -1 & \text{when } x < 0 \end{cases}$$

This definition of the STE function corresponds to the one implemented in Larq [9] in order to binarize the weights and the outputs of the different layers. With this function, the input value x which can be a weight or the output from the previous layer, is fixed to 1 if it is greater or equal to 0 and -1 otherwise. There also exist other functions implemented in Larq but since the STE function gave me the best results, I will only present this one. This affirmation could change depending on the problem addressed.

The gradient of the STE function is defined as such:

$$\frac{\partial q(x)}{\partial x} = \begin{cases} 1 & \text{when } |x| \leq 1 \\ 0 & \text{when } |x| > 1 \end{cases}$$

The value 1 used to compare the absolute value of gradient $|x|$, is called a clip value since it defines the range where the gradients have a value different from 0. The clip value can have a value different than 1, but in this case, I will stick with it for more convenience. The gradients during backpropagation with this function work as follows: if the gradient possesses a value between -1 and 1, then it can be propagated to the previous layer in order to correct the error of the weights, otherwise, the gradient is set to 0. It is used as such in order to keep the gradients between -1 and 1 in order to not worsen the performances of the BNN [1].

2.4 Binary Neural Network

Finally, we reached the main subject of this thesis which focuses on this special type of neural networks. Binary neural networks are important because they use less energy due to their faster inference time but also due to their simpler calculations

using mainly integer values. Using less resources is an important matter if we want to execute large models on low-computing power computers but it is also a good possibility to reduce our energy consumption from an ecological point of view. The difference between a neural network using double-precision values (DPNN) and a Binary neural network (BNN) is how the weights and the output values are represented inside the network. If we take the formula from Section 2.1 for computing the output of a layer and that we apply the STE function, we can binarize the output of a layer. The logic is the same for the weights but in this case, it is only a strict application of the STE function on the weights. Then the equation is modified as follows:

$$Y_{i,l+1} = q\left(\sum_{i=1}^n q(w_{i,l}) * X_{i,l} + w_{0,l}\right)$$

where $X_{i,l}$ is the input value, $q(x)$ is the quantization function,

$w_{i,l}$ is the weight associated to the unit i in layer l and

$w_{0,l}$ corresponds to the bias of the layer l . The bias is never quantized since it is not considered as a weight because it does not multiply any input X .

The resulting network possesses only binarized weights and the outputs from each layer are also binarized with the exception of the last layer of course, otherwise, the inference result would be problematic depending on the problem. This means that we mainly have integer values which are simpler for calculations than double-precision values. The fact that there are only 1 bit values means that we can also reduce the size of the whole model without changing the number of parameters.

But this description is still a bit theoretical because in practical, it is hard to train binarized values in a network. Consequently, the Library Larq [5] that I used in this thesis implements latent weights [10]. The goal is to simplify the optimization of the binarized weights during the backpropagation process of the network in order to reduce the loss. The binarized representations are only used in the forward propagation when the network needs to classify new data. These latent weights make the training process of BNN comparable to a DPNN. Therefore, the weights are updated in the same way during backpropagation than in a DPNN, the only difference in this process are the gradients calculated through the activation function which corresponds to the STE function. Thus, it is indeed simpler to train a BNN with this method since we are actually training a DPNN but the performances of this network are those of a BNN. And once the weights had converged, we can convert these latent weights, which are floating values, to binary weights and then we obtain a network which is finally a full-fledged BNN.

2.5 XGBoost

XGBoost [11] is a type of machine learning model that was already used in past works for classifying bat calls. XGBoost is based on the Random Forest classifier which is an ensemble of decision trees. Using different weak classifiers such as Decision trees in order to build a bigger model is called a Boosting process.

The difference with the Random Forest is that XGBoost searches to optimize an objective function which depends from the loss and from a regularization term. The loss correspond to a custom function which serves to evaluate how the model classify well the data. The regularization term in the other hand helps the model to avoid overfitting by controlling its complexity. This therefore better helps the model to learn from the training data while managing to generalize the objective function it seeks to learn. There are also major differences between XGBoost and Random Forest during training: The sampling of the training data and the reduction of the loss. While Random Forest uses miss-classified data samples from previous Decision trees for training the new ones in order to reduce the whole model error. XGBoost uses the whole training dataset for training each tree. Unlike in Random Forest, XGBoost does not care about the performance of previous trees in order to construct the new one. The Decision trees are then not weighted depending on their performance like in Random Forest.

The Decision trees used in an XGBoost model during training do not work in the same way than in a Random Forest model. They do not predict labels but a residual which corresponds to the prediction error of the whole model. Therefore, an XGBoost model seeks to reduce this residual error when it adds a new tree. The initial tree uses an initial residual which can correspond to the mean of training labels. It is important that the model computes this initial value as starting point which will be further reduced as new trees are added to the model.

XGBoost infers on new data differently than Random Forest. Even if the trees are not weighted in the same way than in a Random Forest model, they are weighted depending on the residuals from the previous tree. Therefore, the initial tree is not weighted but the following ones are indeed weighted. This weight acts as a regularization multiplier on the residual of the associated tree. Thus, the prediction of the model is given by the sum of each tree residuals weighted by their corresponding weight.

2.6 Non Maximum Suppression (NMS)

Non Maximum Suppression is a common process used to filter same events happening at the same time but in different windows. It is mainly used in object detection in image analysis but it can be applied to call detection from audio files. NMS

aims to reduce the number of windows which describe the same event by obtaining only one window with a maximum size. Indeed, we want to obtain one window associated to one call at a given position in the file.

NMS associate a probability to each window in the file. NMS uses two sets corresponding to the set of windows to compare and the set of windows to keep. The algorithm then takes the window with the highest probability and store it in the keeping set. This window is used to filter windows in the comparison set. The windows are filtered depending on their Intersection over Union (IoU) score. If their IoU score is above a certain threshold, they are removed from the comparison set. This IoU score is defined as:

$$\text{IoU} = \frac{A \cap B}{A \cup B}$$

where A and B are different windows

The numerator of the equation corresponds to the intersection of the two windows when the denominator corresponds to their union. After that, all the windows from the comparison set had been tested with the current window from the keeping set, another window with the highest probability from the comparison set is then stored in the keeping set and the whole process is repeated until there is no window left in the comparison set.

Chapter 3

Previous Works

My thesis is a continuation of different past works on the same subject. Different approaches were explored in these works but in this thesis, the approach will be mainly on a multi class classification problem. The novelty addressed in this thesis is the usage of Binary neural networks.

3.1 Detection

The detection approach was the first one in line since it came from the paper of the Bat Detective team [12]. This paper proposed to use a Convolutional neural network in order to detect bat calls for assessing the current populations. But detection is only about the assertion that a call is present or not, the model which is used for detecting calls does not say anything about the species which emit them. This is why they also proposed another approach in order to classify the species but it was not explored in the paper.

3.2 Multi class classification

This approach is the most explored one since it is also the case in this thesis, but it was also the subject of two thesis from past years. The intuition here is that only detecting bat calls is not sufficient since we want to measure their population depending on their species. Thus, we need to use more classes in order to classify the calls depending on their species. Therefore, it is a multi class classification problem. The difficulty of this approach is that we also need to classify the class corresponding to an absence of call and since it is the most represented class, the problem becomes unbalanced. The problem of unbalanced classes is problematic because it can be difficult for a model to predict minority classes since they are not well represented during training. But there exist some methods to avoid this

problem such as using weighted metrics in order to compensate the weight of each class. This means that the minority classes penalized more the model during training than the majority class.

The first thesis that addressed this problem, was the thesis of Maxime Franco and Corrado Lipani [13]. They proposed the version of the Batmen architecture for classifying the different bat species present in Belgium. They were also the first ones to use the Natagora [14] database, which is composed of recordings collected across Belgium.

The second thesis which addressed this problem, was written by Mélanie Beauvois and Lucile Dierckx [15]. They proposed a revised version of the Batmen which is described in Figure 3.1.

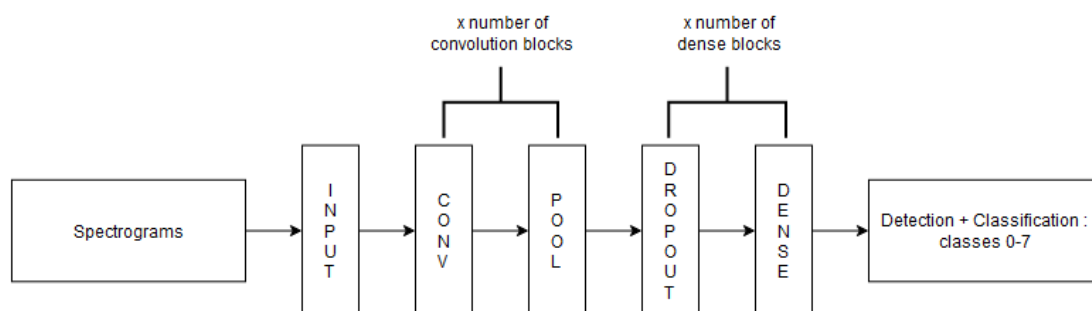


Figure 3.1: Description of the revised Batmen architecture as described in the thesis of Lucile Dierckx and Mélanie Beauvois. The network was composed of 2 Convolution blocks and 2 Dense blocks where the first Convolution layer and the final Dense layer are not counted in the block count.

The architecture from Figure 3.1 is the cornerstone of those that will be discussed in the following Chapters. There is still the exception of ResNet since it is not based on this previous architecture. They also proposed several new architectures of neural networks such as a model using two networks based on the Batmen architecture but also hybrid architectures using other models of machine learning such as XGBoost and Support Vector Machines. These hybrid models also use a network based on Batmen from Figure 3.1 but the last dense layer is dropped in order to use the features computed by the network as input for the other model. They searched to improve the performances of previous architectures such as Batmen but they managed to get the best results using an hybrid model using XGBoost.

3.3 Multi label classification

The last approach was also explored in the thesis of Mélanie Beauvois and Lucile Dierckx [15]. The idea here was to resolve the problem where two calls of different

species overlap which can effectively happen in nature. It means that the model should be able to classify efficiently the calls even if two or more groups are present at the same time. This kind of problem is referred as a multi label classification problem. Thus, they used the Natagora database as well as the databases from the paper of Bat Detective [12] in order to train the same models introduced in their thesis in order to assess their performances. But this time, only a half of the files from the Natgora database can be used for training model since overlapping calls are rarer. Therefore, I hope that the approach from my thesis by using Binary neural networks can be used to explore this domain since they were the first to work on it.

There is also a continuation of their thesis in this paper [16] which was published some months ago. They tried to use data augmentation by creating new data in order to resolve the problem of the limited size of the Natagora database that can be used to train models for a multi label classification problem. They showed that their best model using XGBoost was still able to keep good performances compared to the previous one which does not use data augmentation.

Chapter 4

Datasets descriptions

Since I trained several models on different databases, I will explain their content. There are four different databases that I used to evaluate the different models that I trained in the context of this thesis. Each database stores several audio files containing different bat calls from different sources in Europe:

- Natagora database: The files from this database are provided by Natagora [14] and come from diverse places in Belgium.
- Bulgaria database: The recordings in this database were recorded in Bulgaria but there are also recordings from Romania.
- UK database: Contains mainly audio files recorded across the United Kingdom.
- Norfolk database: Collection of recordings which were specifically recorded in Norfolk.

The last three databases were collected by the Bat Detective [17] Team from the different locations listed above. The Natagora database was shared with the UCLouvain by Natagora [14], thus, the content is private and it cannot be shared outside of the university. Since we are training models, we need to split the different files in two categories named train set and test set. The train set will be used for training the models and the test set will be used to evaluate them. This split gives a certain number of files in each category that can be summarized as follows:

Database	Train set	Test set
Natagora	2306	272
Bulgaria	2812	500
UK	0	434
Norfolk	0	500

Table 4.1: Number of files from each database used for training and testing.

Even if there are four different databases, only the Natagora database can be used for classification since it is the only one to possess files with labels corresponding to the calls of the different bat species. These files were labelled and verified by experts of the Plecotus Team [18] which is the bat studies department of Natagora. Originally, the position of the calls were not specified in the files, but last year, Lucile Dierckx and Mélanie Beauvois used the Bat Detective detection tool in their thesis [15] to detect the positions of the calls inside each file from the Natagora database. Therefore, the files are now labelled with the positions of each call with their corresponding species.

The three other databases contain calls with their position but the species are not specified, thus, they can only be used for binary classification corresponding to the detection task. The Natagora database will be used to train models in a multi class classification problem. The UK and the Norfolk databases can only be used as test sets due to their small size.

We have seen the number of files stored in each database but what about the calls and the species which emit them? The number of calls from each database is summarized in Table 4.2.

Database	Train set	Test set	Total
Natagora	51800	5905	57705
Bulgaria	4782	1604	6386
UK	0	842	842
Norfolk	0	1345	1345

Table 4.2: Number of calls from each database used for training and testing.

As we can observe, the number of calls in the Natagora database is larger than the three others. This is also a reason why I will use it as a training set because it possesses the larger samples of observations. The three others will be mainly used as test sets even if the Bulgaria database can still be used as a training set but the performances might be worse in this case.

Since the Natagora database is the only one with labelled calls, we can take a look at the different numbers of calls with an associated label:

Class	Group	Train set	Test set
1	Barbar	569	109
2	Envsp	15717	1745
3	Myosp	19047	2120
4	Pip35	3916	482
5	Pip50	7959	882
6	Plesp	1188	171
7	Rhisp	3404	396

Table 4.3: Proportions of calls for each label corresponding to a specific group of bat species. The different calls come only from the Natagora database.

The different groups are unbalanced because some species are more common in Belgium than others who are rarer. Therefore, the training process will work accordingly to this distribution and weights will be applied to the classes during training in order to increase the performances. It is mandatory because unbalanced labels can prevent a model from correctly predicting the least present ones.

Concerning the calls properties, they cannot be directly used for training models. Indeed, the audio files contain raw audio which cannot be directly passed as an input through a neural network. The Bat Detective [17] used spectrograms computed directly from the raw audio. These spectrograms can be then used as input. In order to compute the spectrograms, the audio files are decomposed into different windows. Then, an Hanning filter is applied to the audio in order to keep the range of values from the signal in an acceptable range. This filter avoid signal leakage, leakage happens when the power of a signal is scattered on a wider range of frequencies. Therefore, applying an Hanning filter keeps the power of the signal inside the same range but it also modifies the signal such as it begins and ends at 0Hz.

After applying the Hanning filter, a Fast Fourier transform is applied to the signal in order to obtain correct frequency values that could interpreted by a program. But we still need to remove ambient sound which could be problematic during the training process. Therefore, only signal with frequencies between 5kHz and 135 kHz will be kept. These frequencies correspond generally to the range of bat calls, otherwise they will be considered as ambient sounds. Afterwards, the frequencies will be subtracted by their mean amplitude in order to reduce more the effects induced by ambient sounds. Finally, a gaussian filter is applied to the

spectrogram which causes to smooth the whole signal. Therefore, the spectrograms can then be used for training neural networks.

Now that we have discussed about the data that will be used for training the models in the further sections, we need to talk first about different method for binarizing a neural network.

Chapter 5

Methods for binarizing neural networks

This chapter is a first approach for creating neural networks. But before we can train different architectures from previous works, I will present two techniques whose aim to obtain binarized neural networks.

5.1 Network quantization after training

The first method consists in quantizing a neural network after it has been trained. The network possesses double precision weights and the outputs from its hidden layers are floats too. The idea is to binarize the weights and the outputs after the network convergences. In order to achieve this goal, we need to train a neural network. I chose for this task a network which uses the Batmen architecture which was presented in Figure 3.1 but with linear activation function for the hidden layers and with Batch Normalization layers (BN) in place of Dropout layers and after each MaxPool layer. We obtain then this architecture:

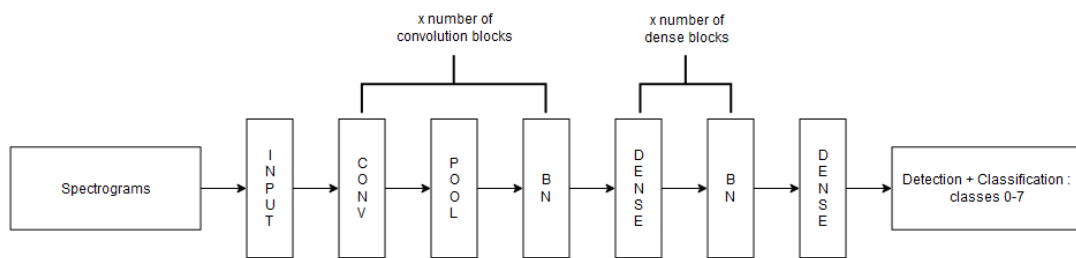


Figure 5.1: First approach to modifying the original Batmen architecture. The additions of BN layers is important in order to obtain good performance for Binarized neural networks.

Afterwards, I quantize their outputs and their weights by applying the STE function.

	No call	Call
No call	1239465	49657
Call	0	8932

Table 5.1: Detection results after network quantization. We can observe the high number of FP which indicates the trouble for the network to detect a call. The precision is 0.15 and the recall is 1 for the Call class.

The Table 5.1 contains the results for the detection of a bat call from the quantized network. These results show the difficulty for the network to discern a call from ambient sound. This is shown by the large majority of FP which makes the predictions for a call irrelevant since the network can not really give a good information about the sound it detects.

	Barbar	Envsp	Myosp	Pip35	Pip50	Plesp	Rhisp
Barbar	0	0	0	0	0	156	0
Envsp	0	0	0	0	0	2633	0
Myosp	0	0	0	0	0	3273	0
Pip35	0	0	0	0	0	739	0
Pip50	0	0	0	0	0	1349	0
Plesp	0	0	0	0	0	171	0
Rhisp	0	0	0	0	0	611	0

Table 5.2: Classification results after network quantization. The fact that only one class is predicted indicates the main problem of using post-quantization in order to create a binarized network. The precision can not be calculated since only one class is predicted but the recall is 0.14 for the whole classification task.

The results for the classification in Table 5.2 confirm this difficulty by the impossibility for the network to differentiate the different classes. This problem is mainly due to the post-quantization since the network can not discern anymore the small differences in the calls between the different species.

There is a possibility to reduce the damages caused by this method on the neural network. It is by constraining the weights and the outputs by taking a range of possible values that is aware of possible results of the quantization function. This works in the case where you want to reduce the number of bits to 8 for representing a weight. But in the case of a binary neural network, this method is not well efficient in post-quantization. Thus, we will use this kernel constraint on the weights during training in order to avoid the problems due to post-quantization.

5.2 Network quantization during training

For training a binary neural network, I used the Larq library [19] which makes this process quite easier since it has been created as an extension for Tensorflow. The main difference for this network which also follows the Batman architecture, is the presence of quantized layers equivalent for the convolution and dense layers. The particularities of these layers is that they use a quantization function for their weights but also as their activation function. To train their weights, there is a constraint applied on them in order to keep their values inside the range of the quantization function domain. As I did previously, the quantization function used to train the network is the STE function.

	No call	Call
No call	1284771	7361
Call	652	5270

Table 5.3: Detection results after training. The structure is the same as in the previous Section, the only differences are the binaries weights and outputs for the hidden layers. The precision is 0.42 and the recall is 0.89 for the Call class.

As we can see in Table 5.3, the results for training the network are better than with the previous method. The precision for detecting a call is higher than before 5.1 since there are less FP, but, it is still worse than tossing a coin. There are also FN in this case, thus it means that the network can fail to detect a call which was not case before.

	Barbar	Envsp	Myosp	Pip35	Pip50	Plesp	Rhisp
Barbar	56	0	45	0	0	0	0
Envsp	5	1128	297	8	12	83	6
Myosp	6	10	1826	10	7	47	5
Pip35	6	72	170	142	14	15	17
Pip50	0	59	377	10	273	55	3
Plesp	0	10	34	2	2	104	0
Rhisp	0	2	81	0	0	0	271

Table 5.4: Classification results after training. The predicted classes are more distributed than in the previous Section. Nevertheless, some of them are still not well predicted. The precision is 0.75 and the recall is 0.62 for the whole classification task.

The results given in Table 5.4 are also better than in Table 5.2. The network can discern more classes than in the post-quantized version. But some classes are

still hard to predict for this network. This is verified by the presence of more or less equivalent number of different predicted classes for a given class, for example, for the Pip35 species.

The performances are still poor but it is only a first approach for comparing these two techniques. We will see afterwards that by modifying a bit the architecture, we can obtain much better results that are closer to the original architectures. This is the reason of the next topic which is about testing the different previous architectures while using quantization layers for training them.

Chapter 6

Comparisons with previous architectures

In this Chapter, we will discuss about the previous architectures which were designed for solving this problem. The comparisons between these architectures will be done by comparing the binary network with its double precision equivalent. Nevertheless, some changes have been done in order to optimize the training of the binary network. Thus, some performances may slightly differ from previous works results.

6.1 Batmen

The architecture used in this Section is based on the improved Batmen architecture [15] and is slightly different from the previous architecture used in the previous Chapter. Even if the basis is the same, there are some modifications that have been made in order to obtain good performances for the BNN. Thus, the architecture has been revised as follow:

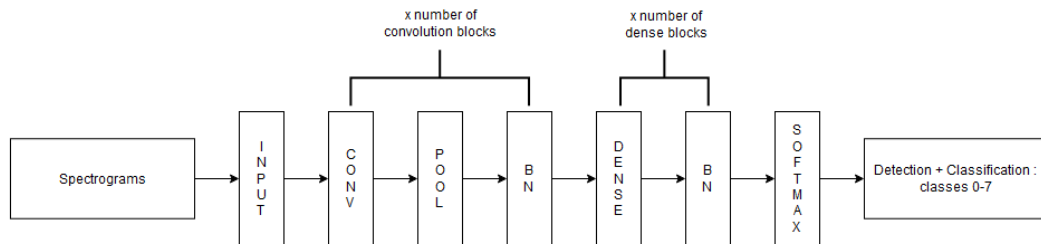


Figure 6.1: Architecture revision of Batmen for training BNNs with good performances. The main differences with the previous architecture are the presence of BN layers and the removal of Dropout layers.

In comparison to the previous Batmen architecture [15], the number of convolution blocks and dense blocks are still the same. This new architecture uses also two blocks for each type. The addition of BN layers is important for training BNNs due to their property to avoid exploding gradients [2]. This property helps BNNs to reach performances close to the state of the art DPNNs.

	Detection		Classification		Global	
	Precision	Recall	Precision	Recall	Precision	Recall
BNN	0.63	0.90	0.80	0.76	0.62	0.73
DPNN	0.68	0.90	0.83	0.78	0.66	0.74
ODPNN	0.78	0.90	0.86	0.76	0.76	0.74

Table 6.1: Performance of the Batmen models with the new architecture and the original one represented by the ODPNN. The scores in Detection are those from the class 1 which corresponds to a detected call. The scores for the classification are a macro-average from the different scores of each classes (1-7). The global scores correspond to the macro-average over all the different classes (0-7).

We can see in Table 6.1 that the new architecture decreases the overall precision compared to the original model. This decrease is more visible in the BNN due to its binarized structure. The DPNN version with the new architecture also suffers from this architecture with a drop of 10% in precision for the detection task. Nevertheless, the overall recall is still constant between the three models.

In summary, the binarization of the original Batmen architecture causes a global drop of 14% in precision but this spectacular decrease comes mainly from the new architecture. If we take a look at the new DPNN, this drop is only of 10%. This means that we can maybe reduce this gap by using a different architecture.

6.2 Double CNN

The double CNN architecture is equivalent in structure to Batmen but the difference lies in the number of networks used by the model. In fact, a first network is used for detecting a call and a second one classify the detected call. Thus, the number of outputs differs with these two networks:

- The network used for detection has two outputs since it is only used for detecting if there is a call or not.
- The network used for classification has seven outputs which corresponds to the seven classes excepted the zero class which corresponds to an absence of call.

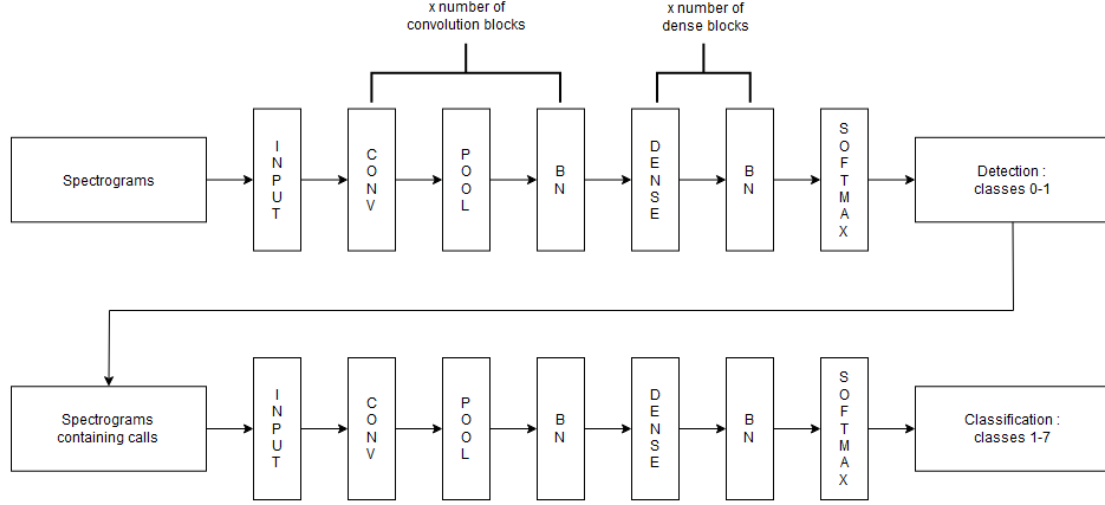


Figure 6.2: CNN2 Architecture using the revised Batman structure.

We can see that the BN layers are in the same positions as in the Batman architecture and there is also a lack of Dropout layers. The number of CNN blocks and Dense blocks are also equivalent to Batman. For now, we will look at the performances for training the CNN2 on the Natagora dataset.

	Detection		Classification		Global	
	Precision	Recall	Precision	Recall	Precision	Recall
BNN	0.79	0.81	0.77	0.69	0.66	0.62
DPNN	0.81	0.81	0.82	0.63	0.70	0.57
ODPNN	0.85	0.84	0.85	0.73	0.77	0.68

Table 6.2: Performances of the CNN2 models with the new architecture and the original one represented by the ODPNN. The scores in Detection are those from the class 1 which corresponds to a detected call. The scores for the classification are a macro-average from the different scores of each classes (1-7). The global scores correspond to the macro-average over all the different classes (0-7).

The results for the CNN2 architecture are better for the detection compared to Batman. But this time, the recall drops more in general compared to the precision for the BNN. This drop in recall is also lower than the DPNN which greatly suffers from the new architecture. Nevertheless, the BNN succeeds to maintain a certain performance for the detection with a loss of 6% for the precision and only 3% for the recall but with the CNN2 architecture, the performances for the classification is worse with a large decrease in performance with the precision. Nevertheless, it is inherent to the CNN2 architecture since this decrease is also observable with

the ODPNN when compared to Table 6.1. Surprisingly, the recall for the BNN is higher than the DPNN, it could be due to the removal of the Dropout layers which greatly help to obtain good performances for DPNN.

6.3 Hybrid model with XGBoost

Now we will discuss about an hybrid model which is composed of a neural network and an XGBoost model as introduced in Section 2.5. The neural network is still based on the Batman architecture but for using the XGBoost model, the second to last Dense layer will be used for training the XGBoost model. Therefore, the architecture can be summarized as follows:

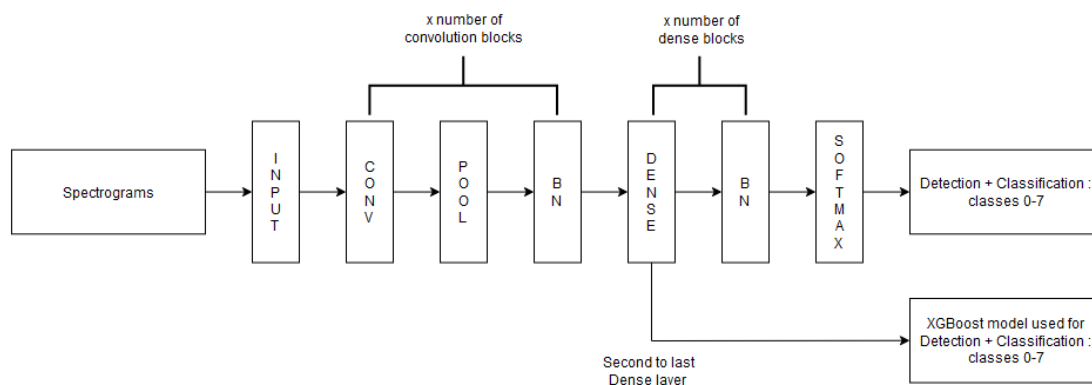


Figure 6.3: The hybrid model Architecture using XGBoost.

As we can see, the removal of the Dropout layers is still in application and there is also the presence of BN layers. The new addition corresponds to the XGBoost model which uses the second to last dense layer in order to make an inference. The XGBoost model is used for detection and classification even if the neural network is trained for both of them. Since the XGBoost model is composed of decision trees, it means that the model is already binarized but the number of trees with a certain depth can leads to several problems which will be covered in another Section. Then, we can take a look at the results of this model with the different types of network used for comparison.

The first observation that we can notice is the closeness of the results between the BNN and the ODPNN which is good news. The only exception is the DPNN which suffers again from a huge loss compared to the ODPNN. This loss is probably due to the new architecture which is not suited for DPNNs. For the BNN, the loss induced by the binarization is greatly compensated by the XGBoost model which reduces this decrease to a maximum of 3% in both precision and recall. This

	Detection		Classification		Global	
	Precision	Recall	Precision	Recall	Precision	Recall
BNN	0.61	0.92	0.83	0.79	0.62	0.76
DPNN	0.51	0.91	0.77	0.72	0.52	0.70
ODPNN	0.60	0.94	0.85	0.81	0.64	0.79

Table 6.3: Performances of the hybrid models with the new architecture and the original one represented by the ODPNN. The scores in Detection are those from the class 1 which corresponds to a detected call. The scores for the classification are a macro-average from the different scores of each classes (1-7). The global scores correspond to the macro-average over all the different classes (0-7).

means that we can use a BNN for inference in combination of an XGBoost model without suffering from a big loss in performance.

6.4 ResNet

We can then discuss about ResNet which stands for Residual Neural Network, this is a type of deep neural network with a lot of layers. It was slightly introduced in the previous Section 2.2 but we will now discussed more precisely about its properties. The particularity of Resnet is the use of several shortcuts between the different convolution blocks. These shortcuts are used in order to keep a maximum of information inside the network and they serve to correct the output of each layer block. They thus allow the network to fit on the residual errors of the different blocks which serve to reduce the risk with vanishing and exploding gradients. Concerning the layer blocks, they are generally composed of several convolution layers with the same number of filters and their padding is equivalent. At the end of these blocks, we find a convolution layer which double the number of filters for the next block but it also downsamples the output with a stride of 2. The architecture used in this paper is based on the ResNet18 as found in the library Larq zoo [8] but this architecture has been modified to make it suitable for a BNN. Since the architecture is different than in the original paper [20], the following table describes the architecture used for training on the dataset.

Block name	Output size	Layer definition
conv1	65x10x64	Conv 64, 7x7, strides 2
pool1	33x5x64	MaxPool, 3x3, strides 2
conv2_x	(33x5x64)x4	Conv 64, 3x3, strides 1
conv3_x	(17x3x128)x3	Conv 128, 3x3, strides 1
conv4_x	(9x2x256)x3	Conv 256, 3x3, strides 1
conv5_x	(5x1x512)x3	Conv 512, 3x3, strides 1
pool2	1x1x512	AvgPool, 5x1, strides 5x1
dense	8	Dense 512, softmax

Table 6.4: Summary of the ResNet architecture as defined in Larq zoo [8]. It is important to know that for using this architecture for BNN, BN layers were added after each convolution block. The downsampling is done after each block.

The ResNet architecture is the biggest one addressed in this paper for a single neural network. The performances showed in the table below only contains the scores from the BNN and the DPNN since there is no modification from a previous architecture.

	Detection		Classification		Global	
	Precision	Recall	Precision	Recall	Precision	Recall
BNN	0.69	0.88	0.82	0.68	0.67	0.64
DPNN	0.68	0.87	0.76	0.63	0.65	0.60

Table 6.5: Performances of the ResNet models. The scores in Detection are those from the class 1 which corresponds to a detected call. The scores for the classification are a macro-average from the different scores of each classes (1-7). The global scores correspond to the macro-average over all the different classes (0-7).

We can see again with this model that the BNN performs better than its equivalent DPNN which means that this architecture is more suited for a BNN. Even if the global precision is the second highest compared to the previous model, the recall is particularly low. This indicates that the ResNet model can not well predict the good class associated to a call. This results with a higher number of FN predicted comparable to the CNN2 architecture. Nevertheless, ResNet still have good performances for detecting call but CNN2 is better in precision.

6.5 Overall Comparison

Now that we have discussed about the different types of architecture, we can compare each of them in order to select a model that will be used for further experimentations.

	Detection		Classification		Global	
	Precision	Recall	Precision	Recall	Precision	Recall
Batmen	0.63	0.90	0.80	0.76	0.62	0.73
CNN2	0.79	0.81	0.77	0.69	0.66	0.62
XGBoost	0.61	0.92	0.83	0.79	0.62	0.76
ResNet	0.69	0.88	0.82	0.68	0.67	0.64

Table 6.6: Performances of the different BNN models. The scores in Detection are those from the class 1 which corresponds to a detected call. The scores for the classification are a macro-average from the different scores of each classes (1-7). The global scores correspond to the macro-average over all the different classes (0-7).

Table 6.6 shows the performance of the BNNs as seen in the previous sections. We can see that Batman does not possess any best score compared to the three others. It is beaten in any task by an other model. But its recall score is still the second best thus it is not the worst model among them.

The other observation that we can see, is the number of best scores obtained by the hybrid model using XGBoost. This is especially true in classification where it possesses the best scores in both precision and recall. The only problem is its precision score for the detection, but this lack of precision might be resolved by the XGBoost model when it is trained on a network with a high precision for the detection like CNN2. The only score where the ResNet model excels, is global precision which is a good thing but as explained earlier, the XGBoost model might be able to compete with it if trained with a good model for detection like CNN2.

In any case, the hybrid model using XGBoost will be used in the further sections for experiencing different methods in order to adapt it to low-computing power computers.

Chapter 7

Architecture optimization for binarized network

The hybrid model using XGBoost, as introduced in Section 6.3, will be executed on different low-computers in a FlatBuffer format which is used for example by the library TensorFlow Lite. The conversion of the current model will be done with the help of the Larq Compute Engine [3], [19] which is a sub-library of Larq build as an extension of TensorFlow Lite for converting specifically BNNs.

In order to optimize the time taken by the processor, we need to change a bit the current the architecture. It is indeed useful because the conversion by the library Larq Compute Engine is the most efficient when some layers are placed at the right positions in the network in order to fuse the different operations efficiently. These problematic layers are in fact the Batch Normalization (BN) layers.

7.1 Moving BatchNorm layers

The problem of BN layers with Larq Compute Engine is the fact that they do not fuse with Pooling layers. Even if the best practice when designing BNNs is to put the BN layers after each Pooling layers, these two operations can not be fused when converted with Larq Compute Engine as explained in Section 2.2. Indeed, the multiplication and the addition operations cannot be applied directly on a MaxPool operation. Thus, they need to be pushed back just after a convolution layer in order to be fused for avoiding unnecessary operations. This is because these two operations can be fused directly since the multiplication operation can directly be applied to the weights of previous Convolution layers as well as the addition operation which acts as a bias, even if there was no bias in the first place. Fusing these operations does not cause any problem on the model size and on the complexity of the calculations since the number of parameters introduced by a BN layer are

much lower than the weights of a DPNN. And as we will see later, a large number of operations are still done with integer values.

Let us take a look at what is happening if we let the BN layers at their current place:

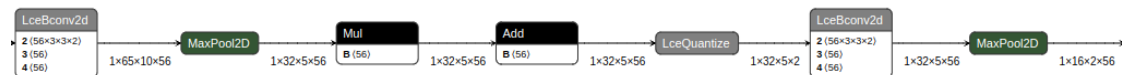


Figure 7.1: Visualization of a non optimized neural network where the BN layers were kept in their current positions. Look at the Mul and Add layers which are the operations linked to a BN layer. The Larq Compute Engine could not fused them due to its position after the MaxPool layer. This image was created with Netron [21].

As explained in the above Figure 7.1, the two operations of the BN layer that were not fused with another layer are problematic because they add more operations for calculating the final output of the BNN. This is indeed problematic because we want to reduce the time taken by the processor for making an inference as much as possible. But, if we look at what happens if we put the BN layers before the MaxPool layers:



Figure 7.2: Optimized version of the same network in Figure 7.1. This time, the two unnecessary layers are gone but they still exist in the same operation in the convolution layer. This image was also created using Netron [21].

When comparing Figures 7.1 and 7.2, we observe that by moving the BN layers, we get only two operations between the convolution layers which is way better than four in the previous case. This modification is also interesting for the MaxPool layer which now uses integer operations. Indeed, we can see from the output dimensions that it is now two channels of dimensions (32x5) which are used to express the resulting tensor from MaxPool layer. It is more efficient to express this tensor with two channels only compared to the 56 channels in Figure 7.1. This reduction with the number of channels comes from the reduction of bits used by the latent weights, which are floating values, from the original model. This reduction is done by the Larq Compute Engine in order to reduce the memory footprint used by the model and the minimum number of 32 bits buffers needed to store 56 bits is 2. So if we want to get an efficient number of buffers, it is mandatory to use a number of

filters which is a multiple of 32, this corresponds to the data type Float32 which is the default Float type used in TensorFlow [6].

7.2 Results of the new architecture

Now we can compare the results obtained between the optimized architecture of the hybrid model and the architecture described in Figure 6.3. The comparison will be done between the BNNs designed with the previous architecture and the optimized one as well as the DPNNs.

	Detection		Classification		Global	
	Precision	Recall	Precision	Recall	Precision	Recall
BNN1	0.61	0.92	0.83	0.79	0.62	0.76
DPNN1	0.51	0.91	0.77	0.72	0.52	0.70
BNN2	0.65	0.89	0.83	0.79	0.62	0.75
DPNN2	0.71	0.92	0.84	0.79	0.69	0.75

Table 7.1: Performances of the different models with the optimized architecture and the previous one. The scores in Detection are those from the class 1 which corresponds to a detected call. The scores for the classification are a macro-average from the different scores of each classes (1-7). The global scores correspond to the macro-average over all the different classes (0-7). The numbers at the end of each acronym stands for the architecture: The number one is for the first architecture and the number two is for the optimized architecture.

As we can see, the optimized architecture gives better results than the previous one for the detection task. It even improves the results obtained with the DPNN2 with a score of 0.71 in precision which is much better compared to the its previous version. The BNN2 also improves in precision but suffers this time from a decrease of 6% in detection and 7% in global compared to the DPNN2. There is also a decrease of 3% in recall for the BNN2 in the detection task, but in global, this decrease is not really important. Thus, this new architecture is more adapted for running on a low-computing machine but it also gives equivalent results to the previous architecture. But there is still a part of this model that we can reduce in size which is the XGBoost model and this what we will do in the next Chapter.

Chapter 8

Reducing XGBoost model size in memory

In order to reduce the size of the XGBoost model, we need to modify some of its parameters. These parameters have an impact on the performances of the model when it has been trained but also on its inference time. Therefore, we will analyze these different parameters in order to measure their impact on the performances but also on the model size.

The performances will be mainly analyzed on the precision scores from the detection task and from the global task. This choice for analyzing the performances is due to the fact that the recall does not change so much with the previous modifications but also with the coming ones. Thus, I won't talk about the classification scores for the same reason.

8.1 Number of estimators

The number of estimators of an XGBoost model corresponds to the number of trees used by the model in order to make an inference. Thus, adding a tree makes the model heavier in memory but it also increases the inference time. The performances will certainly decrease when we reduce too much the number of trees but increasing this number could slightly increase them. Then we need to be cautious when comparing several models because they could have similar performances but their inference time could differ due to this parameter.

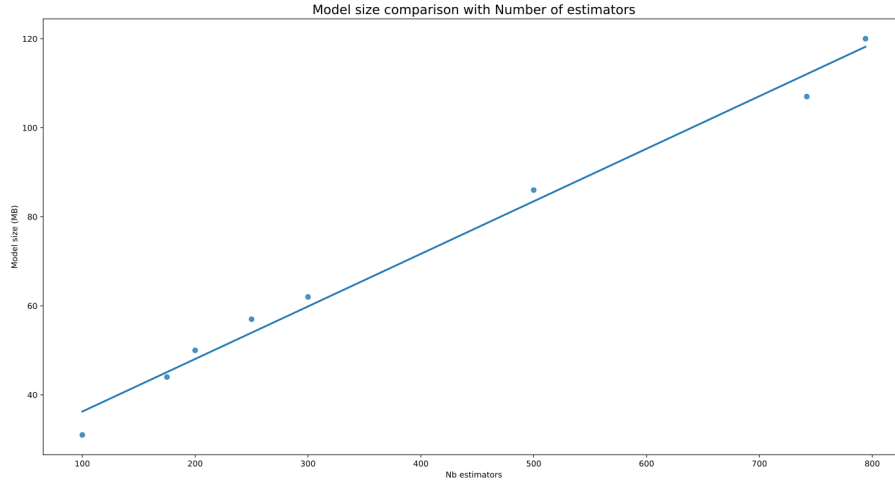


Figure 8.1: Evolution of the model size in memory (MB) compared to its number of estimators.

In Figure 8.1, we can see the relation between the XGBoost model size in MB compared to the number of estimators. This relation increases linearly as the number of estimators grows. But we need to check what happens to the precision value when the number of estimators increases in order to know if it increases the performances or not.

The Figure 8.2 shows the evolution of the precision value for the detection and the global task. In the left Figure depicting the precision value from the detection task, we can see that the relation is linearly dependant, even if some values shows some decrease in performance due to the random process that is the training of a neural network. It is different for the global precision which increases to a maximum of 300 estimators but after this value, the precision is stuck in a plateau. This means that we can choose a number of estimators to a maximum of 300 until we can not increase anymore the global precision. This is rule only holds for a model using a max depth of 10 and a minimum child weight of 3 but this threshold might also be present for higher values which still to be determined.

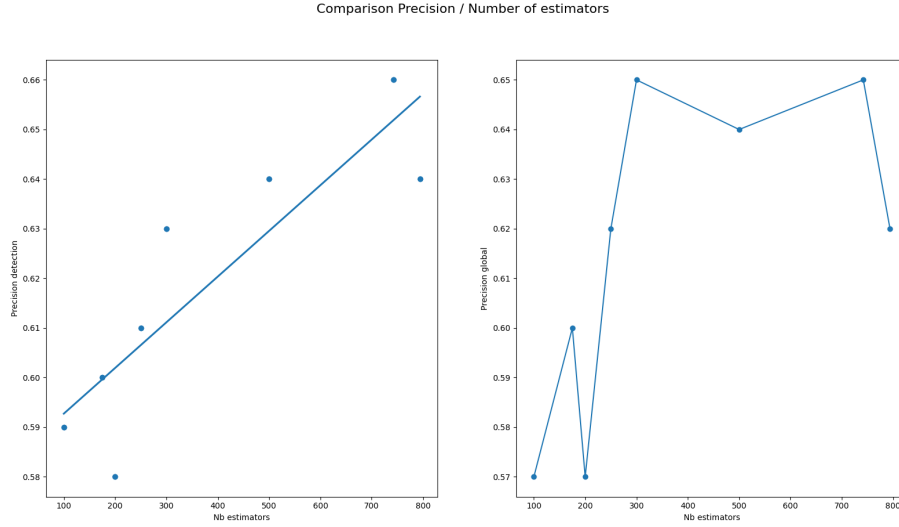


Figure 8.2: Relation between the number of estimators with the Detection (left) and the global precision (right). We can see that the precision for the detection task can still improves from increasing the number of estimators, but, the global precision is stuck on a plateau which means that the precision might not be able to increase further. Maybe it can still increases but the memory space used by the model will also follows this trend. The models possess a maximum depth of 10 and a minimum child weight of 3.

8.2 Maximum depth

Now, we will discuss about an important parameter known as the maximum depth. This parameter put a limit to the height of each tree to the condition that there is still enough samples for splitting a leaf in order to reach this limit. In comparison with the number of estimators parameter which influences directly the number of trees used by the model. The maximum depth can affect the size of each tree, this means that its effect on the model size can be important. Therefore, we can take a look at Figure 8.3 to observe the relation between maximum depth and model size.

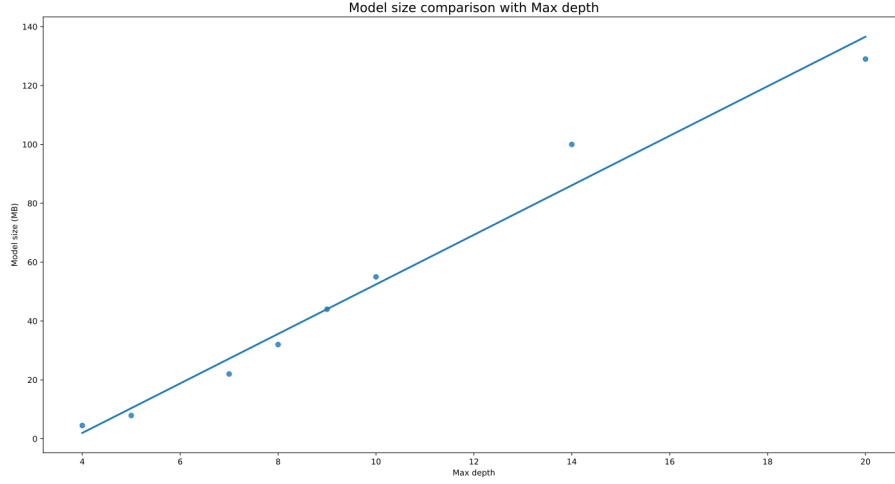


Figure 8.3: Relation between maximum depth and model size (MB).

Equivalently to the relation in Figure 8.1 about the number of estimators. The growth of the model size also follows the increase of the maximum depth. But in this case, it is sufficient to just have a growth of maximum depth from 10 to 14 for doubling the model size. In comparison, doubling the number of estimators indeed multiply by the 2 the model size. This means that the maximum depth has an higher impact on the model size than the previous parameter. Thus, choosing a good value for maximum depth is important for having a good balance between model size and performances.

About the performances, we can see in Figure 8.4 that the precision values follow the same trends as seen in Figure 8.2 but the plateau for the global precision seems to be more stable than in the previous case. Then, increasing the maximum depth value seems to be insignificant beyond a certain point corresponding to a value of 9 or 10. Afterwards, the performances do not increase much more compared to the left side of the graphic showing the evolution of the global precision. This limit might be due to the difficulty for the network to extract good features in order train efficiently the XGBoost model. But as said previously, training specifically a network for an efficient detection, like for the CNN2 architecture, might increases this threshold.

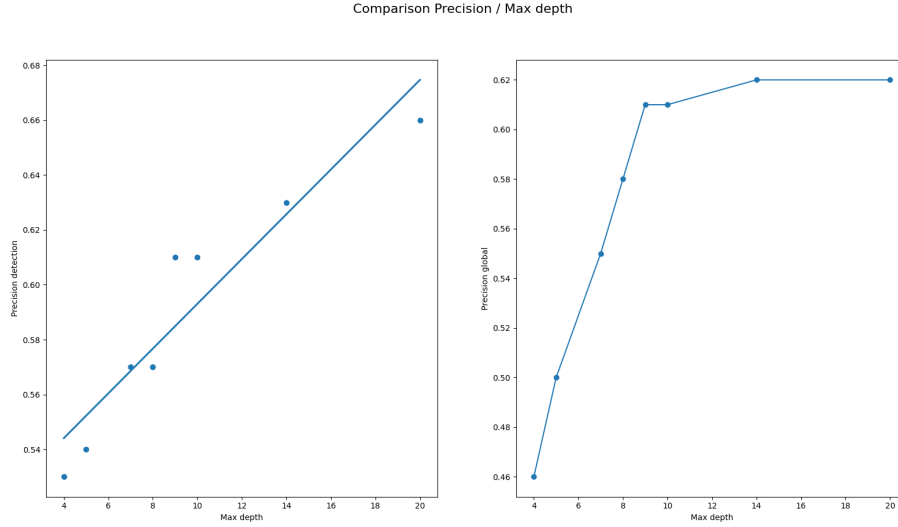


Figure 8.4: Relation between the maximum depth with the Detection (left) and the global precision (right). We can see that the precision for the detection task can also still improves by increasing the maximum depth, but, the global precision is stuck on a plateau which means that the precision might not be able to increase further. Maybe it can still increases but the memory space used by the model will also follows this trend. The trends for the precision values are comparable to the Figure 8.2 but the increase in memory space will be greater in this case. The models possess a number of estimators of 250 and a minimum child weight of 3.

8.3 Minimum child weight

This is the last parameter that can influence the performances of a given model as well as its size. Minimum child weight stands for the minimum number of samples for splitting a leaf in a decision tree. Thus it can avoid the creation of unnecessary leaves leading to a reduction of the model size. This parameter is useful because it tends to reduce the size of the model by keeping the same performances when it is well tuned. Thus, it means that the trend followed by the model size, when increasing the minimum child weight, will be negative in this case when compared to the previous parameters. Let us see if this trend is true by comparing the two values.

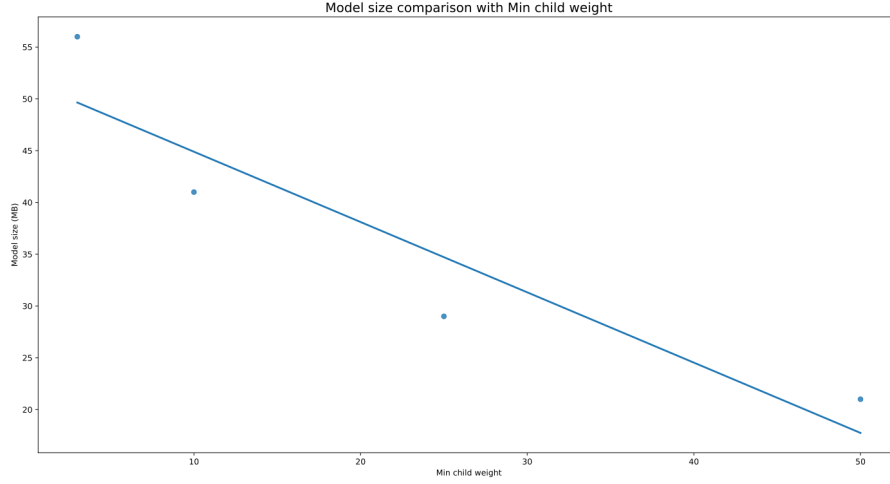


Figure 8.5: Relation between the model size (MB) and the minimum child weight parameter.

As I told before, we can see in Figure 8.5 that the minimum child weight effectively affects the model size negatively. Which means, that it can be used to reduce the size of the model when the other two parameters have been fixed. Nevertheless, increasing this value can not be done without monitoring the performances because this reduction in size can lead to a decrease in performances such as in Figure 8.6.

We can see in Figure 8.6, increasing the minimum child weight parameter leads to a decrease in performance. But, if you look at the values 10 and 25 for the minimum child weight on the Figure 8.6, we can see that the increase in this parameter does not necessarily leads to a drop in performance. The global trend tends to zero but if you look locally, you could see that it decreases by step thus leading to a decrease in model size while keeping equivalent performances. This is not well visible in Figure 8.6 but there are some cases which follow this rule in the next Section.

Now that we have discussed about these three parameters which can affect the performances but also and more importantly the model size. We can use these information to select a model among different possibilities with different values for each parameter.

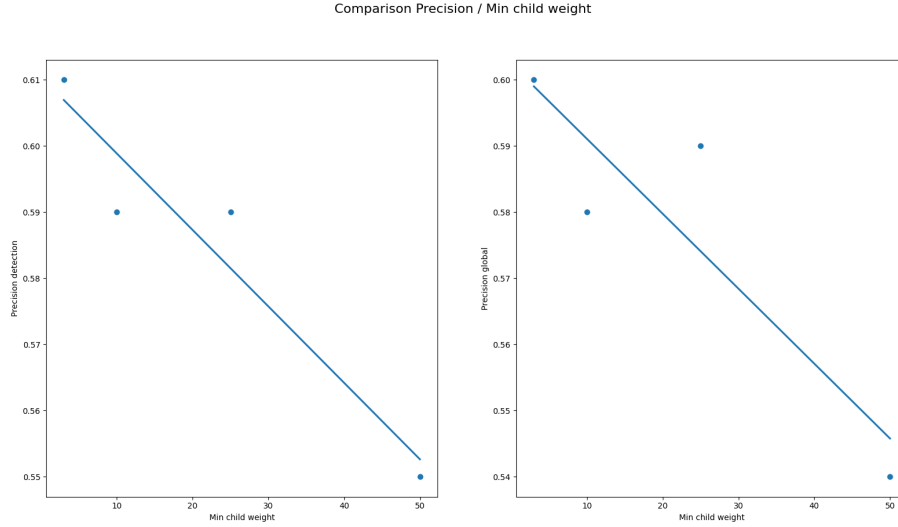


Figure 8.6: Comparisons of the different precision values with the minimum child weight. This time, the effect of this parameter affects negatively the performances when increased. This means that it can be used to reduce the size of a model but, we need to be aware about the performances which could drop. Even if the performances drop by increasing this parameter, they can still be stable if this increase is small while decreasing the model size. We will see a good example further in the next Section.

8.4 Choosing a model

Since we know about the different effects of each parameter during the training process of a XGBoost model. We can now select a model which will minimize the model size while keeping good performances. For comparing each model, I will use different scatterplots representing the relation between their size and their different precision values. The different parameters for defining each model such as their number of estimators, maximum depth or minimum child weight will be used as categorical variables in the different plots in order to distinguish them. Thus, you can identify the parameters for a given model by checking its different colours in each scatterplot. I also want to mention that the relevant models for the selection will be surrounded with a red circle while the original model such as defined in Table 7.1 will be surrounded in black.

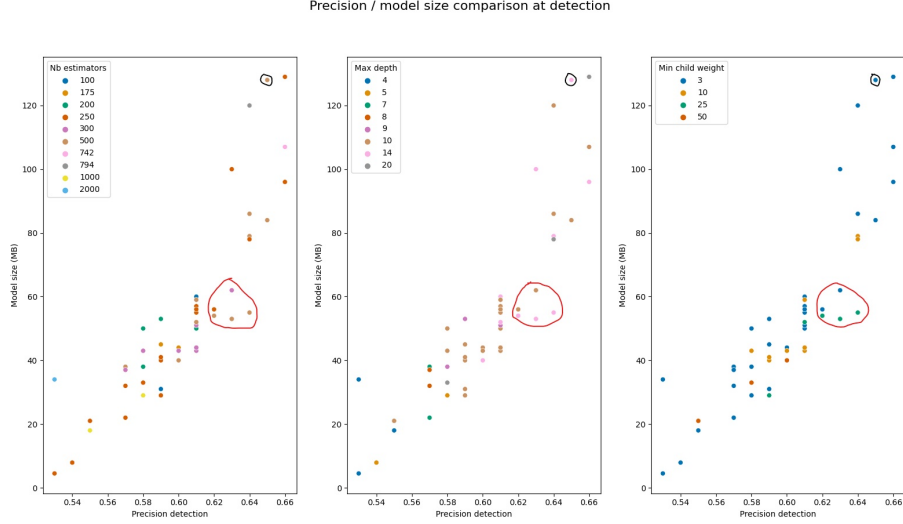


Figure 8.7: Relation between the model size and the precision from the detection task. The parameters for the models are indicated with different colours. The order for the parameters represented in each scatterplot are from left to right: number of estimators, maximum depth and minimum child weight.

As I said, the relevant models which maximize the precision while minimizing the model size are inside the red circle. These models are located in the lower right corner in Figure 8.7 which is understandable since this corner corresponds to the maximum value possible for the precision and the minimum value for the model size. We can see that there are five models in the red circle. Three of them are variants of the original model with a different value for the minimum child weight only which is 25, their other parameters are still the same. Their different values in precision are due to the randomness induced by the training process thus, it can happen that some equivalent models might obtain different precision values. I also would like to point out that in this case, the change of the minimum child weight value induces a drop in model size and not in precision which is related to what I was saying in Section 8.3.

There are also two other models with different parameters compared to the three others. They possess the same value for the maximum depth but their number of estimators are different. One possesses 250 estimators and its model size is lower, the other possesses 300 estimators and is more efficient. Since we have some models to compare, we can now take a look at the global precision.

Concerning the global precision, there are only four models left since the last one remains now in the majority group, it is located slightly to the left of the red circle. The four models are still the same from the scatterplots in Figure 8.7 but

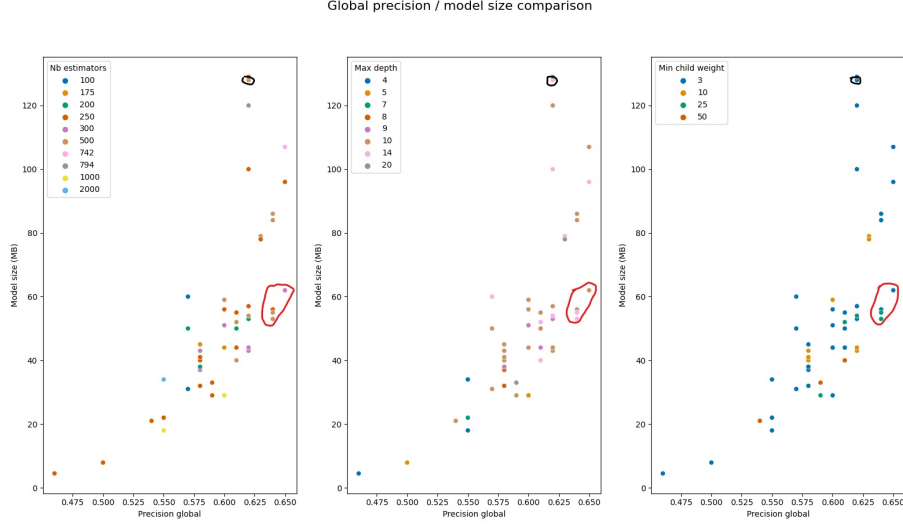


Figure 8.8: Relation between the model size and the global precision. The parameters for the models are indicated with different colours. The order for the parameters represented in each scatterplot are from left to right: number of estimators, maximum depth and minimum child weight.

this time, their precision are more or less equivalent than for the detection task. Three of them form a vertical line which means they possess the same precision but their model size differ. There is only one that performs better but its size is bigger.

Thus, the final model will be chosen among the three models located on the vertical line. Among these three models, there are only two different sets of parameters corresponding to (250, 10, 3) and (500, 14, 25), the order used here for describing the parameters is the same than in the scatterplots from Figure 8.8. The obvious choice would be the second set because it succeeds to reach a minimum model size. Nevertheless, using this set of parameter is problematic due to the number of estimators. Indeed, the number of estimators in the second set is doubled compared to the first one, and the maximum depth is also higher even if the minimum child weight help to compensate its effects on the model size and the growth of each tree. Thus, the model using the first set of values will be much faster than the second one at inference time. This observation is coherent because the maximum number of nodes traversed by a model using the first set is 2500, which is smaller than 7000 for the second set.

For this reason, I will use this model to make inferences on different models of Raspberry Pi. In comparison, this model uses a total of 56 MB of memory in order to be stored compared to 128 MB for the original model.

We can also compare this model with its DPNN equivalent and observe the

differences between the two.

	Detection		Classification		Global	
	Precision	Recall	Precision	Recall	Precision	Recall
BNN	0.62	0.90	0.84	0.79	0.64	0.75
DPNN	0.66	0.92	0.83	0.78	0.66	0.75

Table 8.1: Performance comparison of the final model with its DPNN equivalent. We can see that the difference in performance is closer since the XGBoost model was reduced in order to minimize its size. Thus, we can observe that the accuracy for the detection of BNN is 62% which is 4% lower compared to DPNN. However, at the global level, this difference is only 2%. These differences are much lower than in the original case in Table 7.1 where the difference in detection was 6% and 7% in global.

As we can see from Table 8.1, the BNN performance is more or less equivalent to the DPNN at the global task. There is still a small difference in precision for the detection task but it is still much lower than for the original model BNN2 from Table 7.1. It should be possible to reduce this gap in detection but it would need to maybe change the architecture in order to find a better result or we can imagine to process the input data differently, but it will not be explored further in this thesis. Therefore, we can use this model without losing too much precision compared to the DPNN.

Chapter 9

Executing the final model on a Raspberry Pi

In this Chapter, I will compare the different execution times obtained by the model for different models of Raspberry Pi. The models used for the benchmarks were the model 3B [22] and the model 400 [23] which is comparable to the model 4B using 4GB of RAM [24]. The benchmarks will be divided in two parts for each model of Raspberry Pi depending on the number of threads used for executing the inference of the model. There will be also four different execution times: the feature computation, the detection task, the classification task and finally, the non maximum suppression. The benchmarks use an audio file of 3 seconds resulting in a batch size of 658 possible calls to infer.

	Model 3B		Model 400	
	1 Thread	4 Threads	1 Thread	4 Threads
Feature computation	1.35	1.45	0.34	0.33
Detection	11.84	6.08	4.9	3.03
Classification	0.4	0.44	0.22	0.22
NMS	0.04	0.05	0.02	0.02
Total	13.63	8.02	5.48	3.6

Table 9.1: Benchmarks of the model using a DPNN on the different Raspberry Pi. The detection stands for the network execution because its results are more important for detecting a call. The classification time stands for the execution of the XGBoost model. The number of threads only affects the detection times because the interpreter used for executing the network accepts a custom number of threads and thus I tried to use the minimum and the maximum number of possible threads for each model of Raspberry Pi. The durations are in seconds.

As we can see in Table 9.1, the execution time of the network for making an inference of the whole file is quite huge. In any case, the duration of the DPNN execution exceeds 3 seconds. The second task with the highest execution time is the feature computation which corresponds to applying the spectrogram on the whole file in order to retrieve the features for classifying a possible call. But even if it is the second longest task, it does not exceed 2 seconds for each Raspberry Pi. The XGBoost is way faster than the network since it is only a tree traversal to one of the leaves repeated 250 times which is cheaper in computation compared to the whole execution of a neural network with floating values. The NMS does not take much time because it is only applied on the detected call in order to get the best time window for representing them.

But these execution times are only about the DPNN, now, we want to know in how much time the BNN can achieve its inference.

	Model 3B		Model 400	
	1 Thread	4 Threads	1 Thread	4 Threads
Features computation	1.29	1.29	0.33	0.32
Detection	2.66	1.77	1.06	0.66
Classification	0.34	0.33	0.2	0.2
NMS	0.04	0.04	0.02	0.02
Total	4.33	3.43	1.61	1.2

Table 9.2: Benchmarks of the model using a BNN on the different Raspberry Pi. The detection stands for the network execution because its results are more important for detecting a call. The classification time stands for the execution of the XGBoost model. The number of threads only affects the detection times because the interpreter used for executing the network accepts a custom number of threads and thus I tried to use the minimum and the maximum number of possible threads for each model of Raspberry Pi. The durations are in seconds.

The resulting times for the BNN execution are remarkable compared to the DPNN in Table 9.1. It is even possible for the whole inference to not exceed 1 second for the model 400 with 4 threads as we can see in Table 9.2. The time reduction due by the binarization is of more or less 77% which is spectacular. The other times are comparable since there were no modifications in their cases.

These results show that we can imagine to use a BNN for making real time inference to the condition that we can find a way to reduce the features computation time. But there is another interesting observation before we can conclude, it is the size of the Binary model. The model file stored in a FlatBuffer format has a size of only 14KB which makes it a lightweight compared to the DPNN with a size of

225KB. This reduction in size is approximately 94% with a ratio of 16 if we divide the DPNN size by the size of the BNN. This value is not random because it is caused by the fact that the number of units in each layer are not multiples of 32. As I explained it in a Section 7.1, since there are 56 units in each layer, we need two buffers with a size of 32 bits for storing these 56 units on 1 bit and thus, the size reduction of the whole model can only be a ratio of 16.

Chapter 10

Conclusion

In conclusion, I showed in this thesis that we can binarize a neural network of reference while keeping good performances. By binarizing a network, we can profit of some properties such as better time inference but also a reduction in model size. I also showed how to tune the parameters for another type of machine learning model such as XGBoost in order to reduce its size in memory.

Binary neural networks are interesting because they can keep the structure of a neural network that uses double precision values and still get performances close to the state-of-the-art networks. But even if their performances can slightly drop, their advantages to be faster at inference time and to use less memory make them interesting. Indeed, the fact that they can infer a result in less time than their DPNN equivalent with a great number of integer values, induces that they use less resources from the CPU. Which makes it possible to run them on small devices with less computational power such as a Raspberry Pi. This is an important result if we want to use a small device for capturing bat calls in the nature. Because, if the device use a battery, it will not run out of power slower with this type of network than with a DPNN. And even better, it might be possible for it to infer in real time the species of a bat if it detects a call.

To a certain extent, we can imagine to democratise the usage of BNNs in the industry or in a public usage if it is indeed possible. Due to their less expensive cost in energy usage, they could have a great impact on our consumption of energy which is a good approach when we talk about Green Computing. I think that it is important in our time because the people are concerned about the climate but also about the increasing prices for energetic resources. Therefore, this recent type of neural network might have a great impact in the future, this is only a question of time before it gets the attention that it needs as we have seen in the past ten years about the classical neural network. They possess a spectacular potential because we can use them with just about anything. The only limit is the imagination and the inventiveness of the one who used them.

Chapter 11

References

- [1] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, *Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1*, 2016. DOI: [10.48550/ARXIV.1602.02830](https://arxiv.org/abs/1602.02830). [Online]. Available: <https://arxiv.org/abs/1602.02830>.
- [2] E. Sari, M. Belbahri, and V. P. Nia, *How does batch normalization help binary training?* 2019. DOI: [10.48550/ARXIV.1909.09139](https://arxiv.org/abs/1909.09139). [Online]. Available: <https://arxiv.org/abs/1909.09139>.
- [3] “Webpage of the larq compute engine library.” (), [Online]. Available: <https://docs.larq.dev/compute-engine/>.
- [4] S. Ioffe and C. Szegedy, *Batch normalization: Accelerating deep network training by reducing internal covariate shift*, 2015. DOI: [10.48550/ARXIV.1502.03167](https://arxiv.org/abs/1502.03167). [Online]. Available: <https://arxiv.org/abs/1502.03167>.
- [5] L. Geiger and P. Team, “Larq: An open-source library for training binarized neural networks,” *Journal of Open Source Software*, vol. 5, no. 45, p. 1746, Jan. 2020. DOI: [10.21105/joss.01746](https://doi.org/10.21105/joss.01746). [Online]. Available: <https://doi.org/10.21105/joss.01746>.
- [6] Martín Abadi, Ashish Agarwal, Paul Barham, *et al.*, *TensorFlow: Large-scale machine learning on heterogeneous systems*, Software available from tensorflow.org, 2015. [Online]. Available: <https://www.tensorflow.org/>.
- [7] “Webpage of the larq zoo library.” (), [Online]. Available: <https://docs.larq.dev/zoo/>.
- [8] “Webpage about the resnet implementation in larq zoo.” (), [Online]. Available: <https://docs.larq.dev/zoo/api/literature/#binaryresnetv1>.
- [9] “Webpage about the ste function as implemented in larq.” (), [Online]. Available: <https://docs.larq.dev/larq/api/quantizers/#stesign>.

- [10] K. Helwegen, J. Widdicombe, L. Geiger, Z. Liu, K.-T. Cheng, and R. Nusselder, “Latent weights do not exist: Rethinking binarized neural network optimization,” in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d’Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32, Curran Associates, Inc., 2019. [Online]. Available: <https://proceedings.neurips.cc/paper/2019/file/9ca8c9b0996bbf05ae7753d34667a6fd-Paper.pdf>.
- [11] T. Chen and C. Guestrin, “Xgboost: A scalable tree boosting system,” in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD ’16, San Francisco, California, USA: Association for Computing Machinery, 2016, pp. 785–794, ISBN: 9781450342322. DOI: [10.1145/2939672.2939785](https://doi.org/10.1145/2939672.2939785). [Online]. Available: <https://doi.org/10.1145/2939672.2939785>.
- [12] O. Mac Aodha, R. Gibb, K. E. Barlow, *et al.*, “Bat detective—deep learning tools for bat acoustic signal detection,” *PLOS Computational Biology*, vol. 14, no. 3, pp. 1–19, Mar. 2018. DOI: [10.1371/journal.pcbi.1005995](https://doi.org/10.1371/journal.pcbi.1005995). [Online]. Available: <https://doi.org/10.1371/journal.pcbi.1005995>.
- [13] M. Franco and C. Lipani, “Automated monitoring of bat species in belgium,” Anglais, Ph.D. dissertation, UCL - Ecole polytechnique de Louvain, 2020. [Online]. Available: <http://hdl.handle.net/2078.1/thesis:25168>.
- [14] “Website of natagora.” (), [Online]. Available: <https://www.natagora.be/>.
- [15] M. Beauvois and L. Dierckx, “Automated detection of bat species in belgium,” Anglais, Ph.D. dissertation, UCL - Ecole polytechnique de Louvain, 2021. [Online]. Available: <http://hdl.handle.net/2078.1/thesis:30594>.
- [16] L. Dierckx, M. Beauvois, and S. Nijssen, “Detection and multi-label classification of bats,” in *Advances in Intelligent Data Analysis XX*, T. Bouadi, E. Fromont, and E. Hüllermeier, Eds., Cham: Springer International Publishing, 2022, pp. 53–65, ISBN: 978-3-031-01333-1.
- [17] “Webpage of about bat detective.” (), [Online]. Available: <https://www.batdetective.org/>.
- [18] “Webpage of about the plecotus team.” (), [Online]. Available: <https://plecotus.natagora.be/qui-sommes-nous>.
- [19] T. Bannink, A. Bakhtiari, A. Hillier, *et al.*, “Larq compute engine: Design, benchmark, and deploy state-of-the-art binarized neural networks,” *ArXiv*, vol. abs/2011.09398, 2020.
- [20] K. He, X. Zhang, S. Ren, and J. Sun, *Deep residual learning for image recognition*, 2015. DOI: [10.48550/ARXIV.1512.03385](https://arxiv.org/abs/1512.03385). [Online]. Available: <https://arxiv.org/abs/1512.03385>.

- [21] “Webpage of netron, a good web app for visualizing neural networks.” (), [Online]. Available: <https://netron.app/>.
- [22] “Webpage about the raspberry pi 3 model b with its specifications.” (), [Online]. Available: <https://www.raspberrypi.com/products/raspberry-pi-3-model-b/>.
- [23] “Webpage about the raspberry pi 400 with its specifications.” (), [Online]. Available: <https://www.raspberrypi.com/products/raspberry-pi-400/specifications/>.
- [24] “Webpage about the raspberry pi 4 model b with its specifications.” (), [Online]. Available: <https://www.raspberrypi.com/products/raspberry-pi-4-model-b/specifications/>.
- [25] “Webpage of the git used for this thesis.” (), [Online]. Available: https://github.com/NatDW/Binarized_network_bats_calls.
- [26] “Webpage about the raspberry pi 0 2w with its specifications.” (), [Online]. Available: <https://www.raspberrypi.com/products/raspberry-pi-zero-2-w/>.
- [27] M. Alizadeh, J. Fernández-Marqués, N. D. Lane, and Y. Gal, “A systematic study of binary neural networks’ optimisation,” in *International Conference on Learning Representations*, 2019. [Online]. Available: <https://openreview.net/forum?id=rJfUCoR5KX>.
- [28] J. Bethge, H. Yang, M. Bornstein, and C. Meinel, *Back to simplicity: How to train accurate bnns from scratch?* 2019. DOI: [10.48550/ARXIV.1906.08637](https://arxiv.org/abs/1906.08637). [Online]. Available: <https://arxiv.org/abs/1906.08637>.
- [29] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, *Xnor-net: Imagenet classification using binary convolutional neural networks*, 2016. DOI: [10.48550/ARXIV.1603.05279](https://arxiv.org/abs/1603.05279). [Online]. Available: <https://arxiv.org/abs/1603.05279>.
- [30] H. Qin, R. Gong, X. Liu, X. Bai, J. Song, and N. Sebe, “Binary neural networks: A survey,” *Pattern Recognition*, vol. 105, p. 107 281, Sep. 2020. DOI: [10.1016/j.patcog.2020.107281](https://doi.org/10.1016/j.patcog.2020.107281). [Online]. Available: <https://doi.org/10.1016/j.patcog.2020.107281>.
- [31] “Website of the raspberry pi foundation.” (), [Online]. Available: <https://www.raspberrypi.org/>.

Chapter 12

Hardware and Software Specifications

12.1 Software requirements

If you want to execute the code for training BNNs [25], you need to install these different packages:

- Python 3.6
- CUDA 10.2
- cuDNN 7.6.5
- Cython 0.29.21
- hyperopt 0.2.5
- joblib 1.0.0
- numpy 1.16.4
- pandas 1.1.5
- scikit-image 0.17.2
- scikit-learn 0.24.1
- scikit-multilearn 0.2.0
- scipy 1.4.1
- tensorflow [6] 2.1.0

- xgboost [11] 1.4.0
- larq [5] 0.12.1
- larq-compute-engine [19] 0.6.2
- larq-zoo [7] (optional)

12.2 Hardware specifications

12.2.1 Computer for training the models

Concerning the hardware specifications of the computer for training the different models:

- CentOS 8.1
- GPU: NVIDIA GeForce RTX 2080 SUPER
- CPU: Intel® Core™ i7-9800X CPU @ 3.80GHz
- RAM: 32 GB

For the hardware specifications of each Raspberry Pi model, you can find it by following these links [22], [23], [26] but they are also summarized in the following sections.

12.2.2 Raspberry Pi 3B

- OS: Raspberry OS 64bits
- CPU: Quad Core 1.2GHz Broadcom BCM2837 64bit CPU
- RAM: 1GB

12.2.3 Raspberry Pi 400

- OS: Raspberry OS 64bits
- CPU: Broadcom BCM2711 quad-core Cortex-A72 (ARM v8) 64-bit SoC @ 1.8GHz
- RAM: 4GB LPDDR4-3200

12.2.4 Raspberry Pi Zero 2w

- OS: Raspberry OS 64bits
- CPU: 1GHz quad-core 64-bit Arm Cortex-A53 CPU
- RAM: 512MB SDRAM

Chapter 13

Appendices

13.1 Hyperparameters of each model

13.1.1 Batmen

BNN	Number of convolution layers	2
	Number of Dense layers	2
	Number of filters	56
	Filter size	3
	Pool size	2
	Number of Dense units	384
	BatchNormalization momentum	0.9
	Quantization function	ste_sign
	Weights constraint	weight_clip
	Batch size	320
Adam	Learning rate	0.0014
	Beta 1	0.95
	Beta 2	0.999
	Epsilon	1E-8
Early Stopping	Minimum delta	0.0005
	Patience	20

Table 13.1: Hyperparameters of the Batmen model.

13.1.2 CNN2

BNN for detection (CNN2)	Number of convolution layers	3
	Number of Dense layers	1
	Number of filters	48
	Filter size	3
	Pool size	2
	Number of Dense units	384
	BatchNormalization momentum	0.9
	Quantization function	ste_sign
	Weights constraint	weight_clip
	Batch size	128
Adam	Learning rate	0.00032
	Beta 1	0.95
	Beta 2	0.999
	Epsilon	1E-8
Early Stopping	Minimum delta	0.005
	Patience	15

Table 13.2: Hyperparameters of the BNN used for detection in the CNN2 architecture. It can also be called CNN2 because its output size is 2.

BNN for detection (CNN7)	Number of convolution layers	2
	Number of Dense layers	2
	Number of filters	24
	Filter size	4
	Pool size	2
	Number of Dense units	384
	BatchNormalization momentum	0.9
	Quantization function	ste_sign
	Weights constraint	weight_clip
	Batch size	192
Adam	Learning rate	0.0037
	Beta 1	0.9
	Beta 2	0.999
	Epsilon	1E-8
Early Stopping	Minimum delta	0.00005
	Patience	20

Table 13.3: Hyperparameters of the BNN used for detection in the CNN2 architecture. It can also be called CNN7 because its output size is 7.

13.1.3 Hybrid CNN XGBoost

BNN	Number of convolution layers	2
	Number of Dense layers	2
	Number of filters	56
	Filter size	3
	Pool size	2
	Number of Dense units	384
	BatchNormalization momentum	0.9
	Quantization function	ste_sign
	Weights constraint	weight_clip
	Batch size	320
Adam	Learning rate	0.0014
	Beta 1	0.95
	Beta 2	0.999
	Epsilon	1E-8
Early Stopping	Minimum delta	0.0005
	Patience	20
XGBoost	Number of estimators	250
	Maximum depth	10
	Eta	0.0847
	Gamma xgb	0.01
	Subsample	0.7
	Scale positive weight	1.5
	Minimum child weight	3
	Tree method	gpu_hist

Table 13.4: Hyperparameters of the hybrid XGBoost model.

13.2 Inference times of the Raspberry Pi 400 on a 110 seconds file

	Model 400	
	1 Thread	4 Threads
Features computation	39.94	41.45
Detection	144	92.83
Classification	7.88	7.92
NMS	0.52	0.54
Total	192.34	142.74

Table 13.5: Benchmarks of the same model in Chapter 9 using a DPNN on the Raspberry Pi 400. The detection stands for the network execution because its results are more important for detecting a call. The classification time stands for the execution of the XGBoost model. There are no benchmarks with this file on the model 3B because its 1 GB of RAM is not sufficient for loading the file. The durations are in seconds.

	Model 400	
	1 Thread	4 Threads
Features computation	41.99	43.45
Detection	30.44	20.57
Classification	6.74	6.22
NMS	0.53	0.54
Total	79.7	70.78

Table 13.6: Benchmarks of the same model in Chapter 9 using a BNN on the Raspberry Pi 400. The detection stands for the network execution because its results are more important for detecting a call. The classification time stands for the execution of the XGBoost model. There are no benchmarks with this file on the model 3B because its 1 GB of RAM is not sufficient for loading the file. The durations are in seconds.

13.3 Inference times of a Raspberry Pi Zero 2w on a 3 seconds file

	Model Zero 2w	
	1 Thread	4 Threads
Features computation	2.24	2.38
Detection	13.9	6.81
Classification	OOM	OOM
NMS	None	None

Table 13.7: Benchmarks of the same model in Chapter 9 using a DPNN on the Raspberry Pi Zero 2w [26]. The detection stands for the network execution because its results are more important for detecting a call. The classification time stands for the execution of the XGBoost model. The acronym OOM stands for Out Of Memory, it happened because the RAM memory of this model of Raspberry Pi only have 512 MB which is not enough to load the XGBoost model. The durations are in seconds.

	Model Zero 2w	
	1 Thread	4 Threads
Features computation	2.29	2.23
Detection	3.24	2.19
Classification	OOM	OOM
NMS	None	None

Table 13.8: Benchmarks of the same model in Chapter 9 using a BNN on the Raspberry Pi Zero 2w [26]. The detection stands for the network execution because its results are more important for detecting a call. The classification time stands for the execution of the XGBoost model. The acronym OOM stands for Out Of Memory, it happened because the RAM memory of this model of Raspberry Pi only have 512 MB which is not enough to load the XGBoost model. The durations are in seconds.

13.4 Execution time of the XGBoost model with 500 estimators

File duration	Model 3B	Model 400
3 seconds	1.07	0.57
110 seconds	OOM	18.13

Table 13.9: Benchmarks of the XGBoost model using 500 estimators on the models 3B and 400 of Raspberry Pi. As I told in Section 8.4, the execution time is longer compared to the model using 250 estimators which is an inconvenient since they have the same performances even if this model is lighter than the one with 250 estimators. The OOM event for the Model 3B was explained in Table 13.6. The durations are in seconds.

13.5 Metrics Definition

13.5.1 Precision

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

13.5.2 Recall

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

13.6 Problems encountered

I encountered some problems during the realization of this thesis. These problems are only about softwares but one of them is also about the hardware.

13.6.1 Error when training XGBoost model with large batch size

The first problem was about training the XGBoost model after the BNN model. But I discover that Tensorflow [6] does not free the memory that it used for training the BNN on the GPU. Thus it is important to reduce the initial memory allocated to Tensorflow before training any model.

```
config = tf.compat.v1.ConfigProto()
config.gpu_options.allow_growth = True
session = tf.compat.v1.InteractiveSession(config=config)
```

The Python code sample above is what you can use for training a neural network on the GPU but if you use it, you might encounter a problem if you want to train afterwards an XGBoost model.

Thus you can correct this code by adding the following:

```
config = tf.compat.v1.ConfigProto()
config.gpu_options.allow_growth = True
config.gpu_options.per_process_gpu_memory_fraction = 0.4
session = tf.compat.v1.InteractiveSession(config=config)
```

By adding the third line, this will reduce the total allocated memory of Tensorflow to a given fraction. In this case, Tensorflow will only have access to only 40% of the total memory that was initially allocated. But it can still obtain more memory if needed.

13.6.2 Building Larq Compute Engine on Raspberry Pi

Even if Larq Compute Engine is available on PyPi on the vast majority of platforms such that you can download it with *pip*. It is not possible to download it with *pip* on a Raspberry Pi. You need to build the module manually but it is was though to be built as a C package. Nevertheless, I found a workaround for building the

interpreter used by Larq Compute Engine as a Python module. This interpreter is important for running inferences.

First, you need to follow the building steps as showed on the website [3]. Then, you need to use Bazel to build the target interpreter situated in `/compute-engine/larq_compute_engine/tflite/python/` in the path the whole library. You will then be able to retrieve the binary files in the `Bazel-bin` folder at the root of the library. You just copy and paste the python files from the original library in the folder created by Bazel and you are done. You need to copy them because they are not created again by Bazel during the build.

Another important note is to use a 64 bits OS if your Raspberry Pi use a 64 bits CPU. Otherwise you might encounter some problems since Larq Compute Engine is optimized for 64 bits architecture. But it might be caused by building the library as a Python module in this case which may not have been planned by the developers. I did not have the chance to experiment with a 32 bits CPU to see if it can solve the problem. Anyways, using Larq Compute Engine as a C package can only be beneficial since it will not need anymore Python to run and thus it might be better in terms of inference time.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | www.uclouvain.be/epl